Towards Practical Deterministic Write-All Algorithms

Bogdan S. Chlebus [*] Stefan Dobrev [‡] Dariusz R. Kowalski [¶]
Grzegorz Malewicz [†] Alex Shvartsman [§] Imrich Vrto [||]

ABSTRACT

The problem of performing t tasks on n asynchronous or undependable processors is a basic problem in parallel and distributed computing. We consider an abstraction of this problem called the *Write-All* problem—*using n processors write 1's into all locations of an array of size t*. The most efficient known deterministic asynchronous algorithms for this problem are due to Anderson and Woll. The first class of algorithms has *work* complexity of $O(t \cdot n^{\varepsilon})$, for $n \leq t$ and any $\varepsilon > 0$, and they are the best known for the full range of processors ($n = t$). To schedule the work of the processors, the algorithms use sets of q permutations on $[q]$ ($q \leq n$) that have certain combinatorial properties. Instantiating such an algorithm for a specific ε either requires substantial pre-processing (exponential in $1/\varepsilon^2$) to find the requisite permutations, or imposes a prohibitive constant (exponential in $1/\varepsilon^3$) hidden by the asymptotic analysis. The second class deals with the specific case of $t = n^u$, $u \geq 2$, and these algorithms have work complexity of $O(t \log t)$. They also use sets of permutations with the same combinatorial properties. However instantiating these algorithms requires exponential in n preprocessing to find the permutations. To alleviate this costly instantiation Kanellakis and Shvartsman proposed a simple way of computing the permutation schedules. They conjectured that their construction has the desired properties but they provided no analysis.

In this paper we show, for the first time, an analysis of the properties of the set of permutations proposed by Kanellakis and Shvartsman. Our result is hybrid as it includes analytical and empirical parts. The analytical result covers a subset of the possible adversarial patterns of asynchrony. The empirical results provide strong evidence that our analysis covers the worst case scenario, and we formally state it as a conjecture. We use these results to analyze an algorithm for $t = n^u$ ($u \geq 2$), tasks, that takes advantage of processor slackness and that has work $O(t \log^2 t)$, conditioned on our conjecture. This algorithm requires only $O(n \log n)$ time to instantiate it. Next we study the case for the full range of processors $n \leq t$. We define a family of deterministic asynchronous *Write-All* algorithms with work $O(t \cdot n^{\varepsilon})$ contingent upon our conjecture. We show that our method yields a faster construction of $O(t \cdot n^{\varepsilon})$ *Write-All* algorithms than the method developed by Anderson and Woll. Finally we show that our approach yields more efficient *Write-All* algorithms as compared to the algorithms induced by the constructions of Naor and Roth for the same asymptotic work complexity.

Keywords
Parallel algorithms, scheduling, work, contention of permutations, Write-All

1. INTRODUCTION

The universal success of sequential computing can be attributed to the existence of the ubiquitous RAM model that can be "programmed" using widely accepted high-level notation, such that when a program for an algorithm is compiled for execution on any physical implementations of the RAM, the efficiency of the algorithm is reflected in the performance of the compiled program [33]. The same harmony does not exist in decentralized computing. There is no universally accepted programming paradigm, while there are dissimilar platforms that have different scalability properties, and that dictate a particular style of programming that suits narrow-focus compilers.

[*] Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, Warszawa 02-097, Poland. E-mail: chlebus@mimuw.edu.pl.
Work supported in part by the KBN Grant 8 T11C 036 14.
[†] Department of Computer Science and Engineering, 191 Auditorium Rd., Unit 155, University of Connecticut, Storrs, CT 06269, USA. Email: greg@cse.uconn.edu.
[‡] School of Computer Science, Carleton University, Canada. E-mail: stefan@scs.carleton.ca.
[§] Department of Computer Science and Engineering, 191 Auditorium Rd., Unit 155, University of Connecticut, Storrs, CT 06269, USA and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Email: alex@theory.lcs.mit.edu.
Work supported in part by the NSF CAREER Award CCR-9984778 and by the NSF Grant CCR-9988304.
[¶] Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, Warszawa 02-097, Poland. E-mail: darek@mimuw.edu.pl.
[||] Slovak Academy of Sciences, Bratislava, Slovakia. E-mail: kaifvrto@savba.sk.
Work supported by the VEGA grant No. 2/7007/20 and by DFG grant Hr14/5-1.

Despite the lack of closure on whether there is a parallel analogue of the RAM, one model attracted significant attention in the study of parallel algorithms. This model is the Parallel Random Access Machine or PRAM [11]. PRAM combines the simplicity of RAM with the power of parallelism; it is a convenient model for which numerous efficient algorithms have been developed [10, 12, 21]. However, PRAM makes assumptions that, given the current state of technology, make it difficult for it to be implemented as a scalable architecture. A number of works address and deal with this problem (e.g., [6, 8, 13, 18, 28, 27, 30, 31, 34]). Some approaches preserve PRAM as a model for algorithm designers and provide algorithmic simulations of PRAM algorithms on other platforms. It has been shown that solutions for a particular problem can be used as building blocks in constructing such simulations (e.g., [9, 23, 28, 32]). This problem is known as the *Write-All* problem [19]:

Given a t-element array and n undependable processors, set each element of the array to 1.

Write-All captures and abstracts the computational progress that can be achieved in unit time by t correct synchronous processors. Despite its simplicity, solutions for *Write-All* can be used in constructing more complex robust algorithms and simulations of synchronous parallel algorithms on asynchronous or undependable parallel processors.

In the design of practical parallel programs one needs to ensure good performance and dependability on multiprocessors with unpredictable load patterns. Here a common challenge is to efficiently perform t independent tasks on n processors [17]. Such tasks could be copying a large array, searching a collection of data, or applying a function to all elements of a matrix [14, 15]. In such cases a *Write-All* algorithm can be used with the only change being that the assignment to a particular array element is replaced by a distinct task.

Write-All algorithms can be used iteratively to simulate parallel algorithms formulated for synchronous failure-free processors. This normally requires that the individual steps of the simulated processors are made idempotent (since they may have to be performed multiple times due to failures or asynchrony), and that a linear in the number of processors auxiliary memory is made available (to be used as a "scratchpad" and to store intermediate results). While the former can be solved with the help of an automated tool, e.g., a compiler, the latter requires sophisticated solutions because of the difficulty of (re)using the auxiliary memory due to "late writers" (i.e., processors that are slow and that unknowingly write stale values to memory). Examples of solutions addressing these problems include [2, 3, 22].

The *Write-All* problem has been substantially studied for synchronous failure-prone processors [20], and a number of efficient randomized algorithms have been developed for the asynchronous setting, e.g., [2, 3, 7, 13, 22, 28, 27, 30]. However there is a dearth of deterministic asynchronous algorithms. We measure the efficiency of algorithms in terms of *work* that accounts for all machine instructions executed by the n processors during the computation (this generalizes the processor-time product measure of synchronous parallel algorithms). The first deterministic asynchronous algorithm for *Write-All* is due to Buss *et al.* [4], it is called *algorithm X*. Its worst case work complexity is $O(t \cdot n^{\log 1.5})$. The algorithm uses a binary *progress tree* to direct processors to tasks (array elements), and to balance the loads of the processors. This is done with the help of processor identifiers, where the individual bits of the identifier are used to choose between left and right branches within the progress tree. The main drawback of this algorithm is that the exponent of n is fixed (it is about 0.59), which negatively affects the scalability when n approaches t.

The first algorithm to achieve better work complexity for the case when $t = n$ is due to Anderson and Woll [1]. Its worst case work is $O(t \cdot n^{\varepsilon})$ for $n \leq t$ and $\varepsilon > 0$. We call this algorithm *algorithm AWT*. This algorithm uses q-ary progress trees ($q \leq n$) and a set of q permutations of $[q]$, used in conjunction with processor identifiers to let the processor know in what order to traverse each of the q subtrees of each interior node of the progress tree. It was shown that a set of q permutations that possess the property that if we choose an arbitrary permutation of $[q]$ and we compose it with each of the q permutations in the set, then the *sum* of the number of *left-to-right maxima* (see Knuth vol. 3, p. 13 [26]) of the composed permutations is $O(q \log q)$, yields algorithm AWT with $O(t \cdot n^{\varepsilon})$ work [1]. This sum is called the *contention* of the set of permutations. More generally the $O(t \cdot n^{\varepsilon})$ upper bound is maintained asymptotically if the contention is at most $f(q)$, and for every $\varepsilon > 0$ there exists q such that $\log_q f(q) \leq 1 + \varepsilon$, and q is the constant hidden in the asymptotic analysis.

Several approaches can be used to construct sets of permutations with low contention. When the q permutations are chosen randomly then the property that contention is bounded by $O(q \log q)$ is satisfied with high probability [1]. The challenge is to find such a set of permutations deterministically and efficiently. If this cannot be done expediently, then algorithms that use such permutations are primarily of existential significance. Anderson and Woll [1] show how to search for these permutations, however it takes exponential in q processing time. A different approach is given by Naor and Roth [29]. They show that a set of q permutations with contention $O(q^{1+\varepsilon})$ can be obtained such that each permutation can be computed in time $q \cdot \mathrm{polylog}(q)$. The value of q for which the bound is exponential in $1/\varepsilon^3$, and hence such approach is also impractical. Another method of constructing sets of permutations was proposed by Kanellakis and Shvartsman [20]. Their method generates permutations online and at a constant cost per each permutation element. Individual elements of each permutation are computed by processors locally and independently, making this algorithm practical. They conjectured that the resulting set of permutations has low contention, but provided no analysis.

In some situations it is the case that there are substantially more tasks, than processors. This motivates the study of the case $t = n^u$, $u \geq 2$. Observe that in such a situation there is a trivial algorithm with work $O(t^{1+\frac{1}{u}})$, where each processor does all tasks. There are two existing approaches that yield more efficient *Write-All* than this trivial solution. The first is based on a low contention set of permutations. The other uses a processor collision paradigm. We discuss them in turn.

The first approach uses low contention set of permutations. The method was proposed by Anderson and Woll [1], and we call that solution *algorithm AW*. This *Write-All* algorithm is efficient for the case when $t = n^u$, $u \geq 2$. They show that if the contention of the $q = n$ permutations used by the algorithm is bounded by $O(q \log q)$, then work of AW work is $O(t \log t)$. (Specifically, as shown in Lemma 3.5.3 [20], work of this algorithm is bounded by $O(n^2 + \frac{t}{n} Cont(P))$, where P is the set of n permutations used by AW.) Unfortunately the best known method of finding such a set of permutations requires preprocessing that is exponential in n. To alleviate this shortcoming we show, in this paper, strong evidence that we can *efficiently* instantiate a $O(t \log^2 t)$ work algorithm for the case when $t = n^u$, $u \geq 2$. However our result is based on a conjecture.

The other approach uses a processor collision principle to avoid the need for construction of low contention permutations. This method is due to Groote *at al.* [15]. The authors give a family of algorithms for the case $t = n^u$, $u \geq 2$, with work $O(t^{1+\frac{c}{u \cdot 2^u}})$, for some fixed $c > 0$. The algorithms are provably efficient (specifically the work complexity is $O(t \cdot n^{\log_2 \frac{x+1}{x}})$, where $t = n^{\log_2 x}$). These deterministic asynchronous algorithms do not require any preprocessing and are efficient for the practical applications considered by the authors where $n \ll t$, e.g., when $t = n^2$. (Note that the $O(t^{1+\varepsilon})$ bound on work is worse than the conjectured $O(t \log^2 t)$ bound on work for the algorithm that we present here and support with strong analytical/empirical evidence.)

Contributions. In the paper we present a study of the efficiency of deterministic asynchronous algorithms for the *Write-All* problem: *given a t-element array and n processors, set each element of the array to* 1. We consider two cases. First for the *full* range of processor, $n \leq t$, and the other when $t = n^u$, $u \geq 2$. When considering the full range of processors, and specifically when $n = t$, the best known algorithms have work complexity $O(t^{1+\varepsilon})$ for $\varepsilon > 0$, which hinges on the ability to construct, deterministically and efficiently, sets of permutation with low *contention* [1]. Prior solutions relied on expensive pre-processing or yielded solutions with prohibitive constant hidden by asymptotic analysis.

Here we present, for the first time, an analysis of the contention of the set of permutations [20] that are easily constructible by the processors, locally and at the *constant time* cost per element. Our analytical result covers only a subset of the possible adversarial patterns of asynchrony. However, we provide strong empirical evidence that our analysis indeed identifies the worst case contention for these permutations. We formally state a conjecture about this worst case contention. Assuming the conjecture, we instantiate representatives of $O(t \cdot n^{\varepsilon})$-work algorithms and we compare the costs of instantiating such algorithms. We also study the case when $t = n^u$, $u \geq 2$. We show that assuming our conjecture we can efficiently instantiate a $O(t \log^2 t)$ work algorithm for asynchronous *Write-All* problem. This result improves upon the algorithm of Anderson and Woll [1] that requires exponential in n preprocessing. Additional details of our contributions are as follows.

- **Contention of an efficiently computed set of permutations.** We study the contention of the set of q permutations (where $q + 1$ is a prime) proposed by Kanellakis and Shvartsman [20]. We show, in Theorem 3.1 and 3.2, that the contention of the set with respect to a certain family of adversarial permutations is $\frac{3}{\pi^2} q (\ln q)^2 + O(q \ln q (\ln \ln q)^2) = O(q \log^2 q)$. Our exhaustive search shows that the contention of the set with respect to any permutation, for $q + 1$ prime, $q \leq 12$, is never greater and is bounded by the same quantity. A simulation study supports our Conjecture 3.1 that this bound holds for any q, $q + 1$ prime.

We apply this results to construct *Write-All* algorithms for the case when $t \geq n$, as follows:

- **Faster construction of $O(tn^{\varepsilon})$-work *Write-All* algorithms.** We define new families of algorithms $\mathrm{AWTD}_{\varepsilon}$, for any $\varepsilon > 0$. The family $\mathrm{AWTD}_{\varepsilon}$ includes algorithms with $O(tn^{\varepsilon})$ work based on algorithm AWT [1] and predicated on our conjecture. Let $\mathrm{AWTP}_{\varepsilon}$, for a given ε, be the family of of $O(t \cdot n^{\varepsilon})$-work AWT algorithms using the permutations of Anderson and Woll [1]. For any $\varepsilon > 0$, we compare the preprocessing time of algorithms in $\mathrm{AWTD}_{\varepsilon}$ with the preprocessing of the algorithms from $\mathrm{AWTP}_{\varepsilon}$. If we assume that prime numbers are readily available, our algorithms in $\mathrm{AWTD}_{\varepsilon}$ require no preprocessing. Even if primes need to be constructed, we show that algorithms in $\mathrm{AWTD}_{\varepsilon}$ need significantly less preprocessing than algorithms in $\mathrm{AWTP}_{\varepsilon}$ to achieve the same work complexity, this appears in Theorem 4.5.
- **More practical algorithms with the same asymptotic work complexity.** We contrast the efficiency of algorithms from our new family $\mathrm{AWTD}_{\varepsilon}$ with the efficiency of algorithms that use the construction of Naor and Roth [29]. Let $\mathrm{AWTN}_{\varepsilon}$, for a given ε, be the family of $O(t \cdot n^{\varepsilon})$-work algorithms using the permutations as defined in [29]. Algorithms in each family achieve $O(tn^{\varepsilon})$ asymptotic work complexity. We show that an algorithm in our family $\mathrm{AWTD}_{\varepsilon}$ has significantly smaller constant hidden by the asymptotic analysis than any algorithm in $\mathrm{AWTN}_{\varepsilon}$. This appears in Theorem 4.7

Finally we apply the results on low-contention permutations to the case when $t = n^u$, $u \geq 2$, as follows:

- **Analysis of algorithm Y.** We give an analysis of work of algorithm Y [20]. Conditioned on Conjecture 3.1 and supported by empirical evidence, the work of algorithm Y is bounded by $O(t \log^2 t)$ for n processors and $t = n^u$ tasks, $u \geq 2$, this appears in Theorem 4.2. The work of algorithm Y is greater by the factor of $\log t$ than the work of algorithm AW of Anderson and Woll [1], however its preprocessing time is significantly lower. Specifically, the preprocessing for algorithm Y is sublinear in t, see Theorem 4.2, while the preprocessing for algorithm AW is exponential.

The rest of the paper is structured as follows. In Section 2 we give basic definitions and describe the (known) algorithms that are the targets of our analysis and comparison. In Section 3 we provide our analysis of and supporting empirical data for efficient construction of low contention permutations. In Section 4 we use such permutations to specify and analyze efficient algorithms, and we compare the achieved results with prior work. We conclude in Section 5. Due to the space limitation some simpler proofs and proofs of mainly mathematical facts are omitted.

2. DEFINITIONS AND ALGORITHMS

We define the model of computation, work complexity, contention, and describe algorithms.

2.1 Model of computation, and complexity

In our parallel setting we consider a system consisting of n processors with unique identifiers from $[n] = \{1, \dots, n\}$. A processor's activity is governed by its local clock. We model asynchrony (and failures) as an adversary that introduces delays (possibly infinite) between local clock ticks.

We study the complexity of *Write-All* measured as *work* that accounts for all steps taken by the processors. We assume that it takes a unit of time for a processor to perform a unit of work such as writing to a shared memory cell, according to its local clock. Idling processors consume a unit of work per idling step. For an execution $\mathcal{E}$ of an algorithm denote by $n_i(\mathcal{E})$ the number of processors completing a unit of work at time step i of the computation (according to some global time that is not available to the processors). Let $\tau_{\mathcal{E}}$ be the time step when the execution terminates.

DEFINITION 2.1. *For a n-processor algorithm that solves a given problem of size t the work complexity W is:* $W_{t,n} = \max_{\forall \mathcal{E}} \left\{ \sum_{i \le \tau_{\mathcal{E}}} n_i(\mathcal{E}) \right\}$.

2.2 Contention of permutations

Let a *schedule* be a list of task identifiers (not necessarily distinct) from the set $[i] = \{1, \dots, i\}$. For a schedule $\rho = \langle r_1, \dots, r_s \rangle$ a *left-to-right maximum* [26] is an element r_j of the list that is larger than all of its predecessors (i.e., $r_j > r_1, r_j > r_2, \dots, r_j > r_{j-1}$). For a permutation $\rho = \langle r_1, \dots, r_s \rangle$ we denote element r_j by ρ_j.

We define contention for schedules that are permutations. For a schedule ρ that is a permutation on $[i]$ and a permutation δ on $[i]$, the *contention* of ρ with respect to δ denoted as $Cont(\rho, \delta)$ is defined as the number of left-to-right maxima in the schedule $\delta^{-1}\rho$. For a list $R = \langle \rho^1, \dots, \rho^m \rangle$ of schedules that are permutations on $[i]$ and a permutation δ on $[i]$, the contention of R with respect to δ is defined as $Cont(R, \delta) = \sum_{u=1}^{m} Cont(\rho^u, \delta)$. The contention of the list of schedules R is defined as $Cont(R) = max_{\delta \text{ on } [i]} Cont(R, \delta)$.

2.3 Lists of permutations with low contention

In this section we report two families of permutations with low contention. The first family was introduced by [1]. They proved that that for all $n \ge 0$ there exists a list $P = \langle \pi^1, \dots, \pi^n \rangle$ of n permutations with contention at most $3nH_n = \Theta(n \log n)$, where H_n is the n^{th} harmonic number. For a given n a list P is instantiated [1] using exhaustive search. We test every set of n permutations by evaluating the set's contention with respect to all permutations on $[n]$. Hence it takes $O(\binom{n!}{n} \cdot n! \cdot n^2 \log n)$ time to find P. The second family was introduced by [29]. For a given m they take a Kronecker product of all $(m!)^m$ $m \times m$ arrays the rows of which contain the permutations. The resulting list $N = \langle \nu^1, \dots, \nu^n \rangle$ of $n = m^{(m!)^m}$ permutations on $[n]$ was shown to have contention at most $n^{1+\frac{\ln 3 + \ln \ln m}{\ln m} + \gamma_m} = O(n^{1+\varepsilon(m)})$, where γ_m tends to zero at least as fast as $1/\log m$. For a given admissible n the list N can be build quickly [29]. Establishing any coordinate of any permutation takes time which is polylogarythmic in n.

```
01    for all processors j := 1, ..., n do in parallel
02        for i := 1 to n
03            x := ρ^j_i
04            if b_x = 0
05                write to all cells in B_x
06                b_x := 1
```

Figure 1: The Block-Write algorithm. Processor j writes to blocks according to permutation ρ^j.

2.4 Block-Write algorithm and algorithm AW

The Block-Write algorithm for *Write-All* was introduced in [1] (see Figure 1). There are n processors and t memory cells ($t \ge n$). The memory cells are divided into n consecutive blocks, $B_1, \dots, B_n$, each of size at most $\lceil \frac{t}{n} \rceil$. A processor j is assigned a permutation ρ^j of $[n]$, which defines the sequence in which the processor attempts to write the blocks. The algorithm uses an auxiliary table of n *completion bits* denoted as $b_1, \dots, b_n$. Before a processor writes to a block B_x it reads (line 04) the completion bit b_x to see if this block has already been written. If b_x is 1 then the processor does not write to B_x. Otherwise (lines 05-06) it writes to all cells in B_x and sets b_x to 1, and we call these events together a *successful block write*. The efficiency of the algorithm depends on the choice of permutations $R = \langle \rho^1, \dots, \rho^n \rangle$.

LEMMA 2.1. *[1] Let $R = \langle \rho^1, \dots, \rho^n \rangle$ be a list of permutations on $[n]$. For an execution of the Block-Write algorithm, let processor j perform its block writes in the order $B_{\rho^j_1}, \dots, B_{\rho^j_n}$. Then the number of successful block writes is at most $Cont(R)$, and the worst case work of the algorithm is $O(\frac{t}{n} \cdot Cont(R) + n^2)$.*

We call the instance of the Block-Write algorithm for the set of permutations P *algorithm AW*. Algorithm AW solves *Write-All* for n processors and $t = n^u$ cells, $u \ge 2$, with $O(t \log t)$ work.

2.5 Algorithm Y

We now define another instance of the Block-Write algorithm, called *algorithm Y* [20]. First we define a list of schedules.

DEFINITION 2.2. *For a given $k > 0$ we define $D_k = \langle \delta^1, \ldots, \delta^{k-1} \rangle$ to be a list of $k-1$ schedules. Each schedule δ^j of length $k-1$ is defied as $\delta^j(i) = ji \bmod k$, $i = 1, \ldots, k-1$.*

Standard algebraic facts yield that if p is prime then D_p is a set of $p-1$ permutations, and:

LEMMA 2.2. *D_p is a subgroup of S_p, the symmetric group.*

In algorithm Y there are n processors for inputs of size $t = n^u$, $u \geq 2$, where $n+1$ is p, a prime (standard padding techniques are readily used when the number of processors is not just inferior to a prime). Each processor j uses the permutation δ^j of D_p (i.e., $\rho^j = \delta^j$). The key fact is that the permutations need not be pre-computed, as any item can be computed directly by any processor j with the knowledge of the number of work elements w it has processed thus far, as $j \cdot (w+1) \bmod p$.

2.6 Algorithm AWT

Algorithm AWT [1] in Figure 2 is a generalization of algorithm X [20]. There are $n = q^h$ processors, $h \geq 1$, and $t \geq n$ tasks. Each processor has a unique identifier, which is a string of length h over alphabet $[q]$. Each such string w is canonically assigned a distinct number $val(w) \in [n]$. The algorithm is parameterized by a list of q permutations $R = \langle \rho^1, \ldots, \rho^q \rangle$ on $[q]$, and we write AWT(R) when we refer to the instance of algorithm AWT for a given list of permutations R. This list is available to all processors. Processors have also access to $\lceil \frac{t}{n} \rceil$ shared q-ary trees called *progress trees*.

The algorithm starts by each processor calling procedure AWT(). Each processor runs in $\lceil \frac{t}{n} \rceil$ asynchronous iterations (i.e., processors do not wait for each other) When a processor is in iteration number r it traverses the q-ary progress tree number r. Traversing is accomplished by calling the recursive procedure Traverse($r, 0, \lambda$) (line 03). Nodes of the tree are labeled with strings over alphabet $[q]$. A string labeling a node identifies the path from the root to the node (e.g., the root is labeled by λ, the leftmost child of the root is labeled by 1). Each node w of the tree, apart from the root, contains a *completion bit*, denoted by b^r_w, initially set to 0. The leaves of the tree contain the (assignment) tasks to be performed. Processors start at the root (level 0) and traverse the tree. When a processor visits a node on level l of the tree it takes the $(l+1)^{th}$ letter j of its identifier (line 10) and attempts to visit the children in the order established by permutation ρ^j.

The visit to a child $a \in [q]$ succeeds only if the completion bit $b^r_{w \cdot a}$ for this child is still 0 at the time of the attempt (line 13). When a processor visits a leaf it performs the task associated with the leaf (lines 07-08). When a processor has performed all iterations all tasks are guaranteed to be done (see [20], page 94), and the processor can halt (line 04).

The following lemma, due to [1] (as presented in [20]), shows how the work of algorithm AWT depends on the contention of the list of permutations E used by the algorithm.

```
01 AWT()
02     for r := 1 to ⌈t/n⌉
03         Traverse(r, 0, λ)
04     indicate completion and halt
05 Traverse(r, l, w)
06     if l = h then
07         x := (r − 1)⌈t/n⌉ + val(w)
08         if x ≤ t then do task number x
09     else
10         j := q_{l+1}
11         for i := 1 to q
12             a := ρ^j_i
13             if b^r_{w·a} = 0 then
14                 Traverse(r, l + 1, w · a)
15                 b^r_{w·a} := 1
```

Figure 2: Algorithm AWT for the processor with id $\langle q_1 \ldots q_h \rangle$, using permutations $R = \langle \rho^1, \ldots, \rho^n \rangle$.

LEMMA 2.3. *Algorithm AWT for $n = q^h$ processors, $h \geq 1$, and $t \geq n$ tasks using a list R of q permutations on $[q]$ solves the Write-All problem with work $O(q \cdot t \cdot n^{-1+\log_q Cont(R)})$.*

2.7 Families $\mathrm{AWTP}_\varepsilon$ and $\mathrm{AWTN}_\varepsilon$ of $O(tn^\varepsilon)$ work algorithms

In this section we define two families of *Write-All* algorithms. These families are indexed by $\varepsilon > 0$. Each algorithm from the family AWTP_ϵ and the family AWTN_ϵ has $O(t \cdot n^\varepsilon)$ work complexity.

We define the family $\mathrm{AWTP}_\varepsilon$, $\varepsilon > 0$ as:

$$\mathrm{AWTP}_\varepsilon = \{\mathrm{AWT}(R) : q > 0, R = \langle \rho^1, \ldots, \rho^q \rangle, \\ \rho^i \text{ is a permutation on } [q], -1 + \log_q Cont(R) \leq \varepsilon\}.$$

It is known that for every q there exists a set P of q permutations on $[q]$ with contention $O(q \log q)$ [1]. Since $\lim_{q \to \infty} \log_q(q \log q) = 1$, the family $\mathrm{AWTP}_\varepsilon$ is nonempty for any $\varepsilon > 0$. A member of the family may be instantiated using the procedure described in [1]. For a given $\varepsilon > 0$, we pick q large enough so that $\log_q(3qH_q) \leq 1 + \varepsilon$. Then we find a set P of q permutations on $[q]$ by a brute force search.

We define the family $\mathrm{AWTN}_\varepsilon$, $\varepsilon > 0$ using the definition of list N introduced in Section 2.3 as:

$$\mathrm{AWTN}_\varepsilon = \{\mathrm{AWT}(N) : N = \langle \nu^1, \ldots, \nu^q \rangle, \\ -1 + \log_q Cont(N) \leq \varepsilon\}.$$

It is known that for every $\varepsilon > 0$ there exists a list N of q permutations with contention $q^{1+\varepsilon}$ [29]. Hence the family is nonempty.

By Lemma 2.3 any algorithm in $\mathrm{AWTP}_\varepsilon$ or $\mathrm{AWTN}_\varepsilon$ solves *Write-All* in $O(tn^\varepsilon)$ work. Note that a particular member, AWT(R) or AWT(N), of the families solves *Write-All* for $t \geq n \geq q$, where $q = |R|$ or $q = |N|$. Hence the values of n for which a particular member of the family is suited for, and the constant q hidden in the asymptotic analysis may be different across members.

3. EFFICIENT CONSTRUCTION OF LOW-CONTENTION PERMUTATIONS

In this section we first show an analytical bound on the contention of D_p, p prime, with respect to a family of adversarial permutations. Next we show the results of our experimental study that lead us to the conjecture that the analytical bound established on the contention of D_p holds with respect to any permutation. The experimental results give strong evidence in support of the conjecture.

3.1 Analytical bound

In this section we show a bound on the contention of D_p. We start by showing an upper bound on the number of left-to-right maxima for D_k, for any $k > 0$.

Let $L(k,i)$ denote the number of left-to-right maxima in the schedule δ^i of D_k. Let $L(k)$ be the number of left to right maxima in all the schedules of D_k, i.e., $L(k) = \sum_{i=1}^{k-1} L(k,i)$. The next result shows the growth of $L(k)$.

THEOREM 3.1. *For any* $k > 0$,

$$L(k) = \frac{3}{\pi^2} k (\ln k)^2 + O(k \ln k (\ln \ln k)^2).$$

PROOF. The symmetricity of the schedules implies that $L(k, k-i)$ determines the number of the *right-to-left* maxima in schedule δ^i. Define $M(k,i) = L(k,i) + L(k,k-i)$ as the number of *maxima* in the schedule δ^i. Then

$$\sum_{i=1}^{k-1} L(k,i) = \frac{1}{2} \sum_{i=1}^{k-1} M(k,i).$$

We claim that

$$M(k,i) = \sum_{l=0}^{a-1} \left\lfloor \frac{k_l}{k_{l+1}} \right\rfloor, \tag{1}$$

where $k_0 = k, k_1 = i$ and $k_2, k_3, \ldots, k_a$ is the sequence of nonzero remainders in the Euclid's algorithm for the GCD of k_0 and k_1. This claim will immediately imply the theorem, as Knuth mentions [25] (exercise 4.5.3. No. 35 and [35]) that summing up the right hand sides of (1) for $i = 1, 2, 3, ..., k-1$ yields

$$\frac{6}{\pi^2} k (\ln k)^2 + O(k \ln k (\ln \ln k)^2).$$

Proof of the claim. We show that if $k = ui + r, 0 \le r < i$ then

$$M(k,i) = \left\lfloor \frac{k}{i} \right\rfloor + M(i,r), \tag{2}$$

where we assume by the definition that $M(x,0) = 0$, for $x > 1$. Clearly, (1) is a solution of (2).

Case 1: $gcd(k,i) = 1$. Each of the numbers $i, 2i, 3i, ..., (\lfloor k/i \rfloor - 1)i$ contributes by 1 to $M(k,i)$. The last number of schedule δ^i, equal to $\delta^i(k-1) = k-i$, contributes by 1 to $M(k,i)$. It is easy to see that from the remaining numbers of the schedule only those number can contribute to $M(k,i)$ which belong to the interval $[k-i+1, k-1]$. These $i-1$ numbers are the maximum elements in each of $i-1$ runs (an ascending subsequence of the schedule). Using the assumption that k and i are relatively prime, the numbers form a sequence Q expressed in the following way from left:

$$\left\lfloor \frac{bk}{i} \right\rfloor i - (b-1)k, \text{ for } b = 1, 2, \ldots, i-1.$$

Clearly, $M(k,i)$ equals $\lfloor k/i \rfloor$ plus the number of maxima of the sequence Q. Subtracting $k - i$ from each element of Q we get a sequence Q' of numbers from the interval $[1, i-1]$ expressed as

$$\left\lfloor \frac{bk}{i} \right\rfloor i - bk + i, \text{ for } b = 1, 2, \ldots, i-1.$$

Note that the number of maxima of the sequences Q and Q' is the same. Taking the elements of Q' modulo i does not change their values, and, for $b = 1, 2, \ldots, i-1$, we get

$$\left\lfloor \frac{bk}{i} \right\rfloor i - bk + i \bmod i \;=\; -b(ui+r) \bmod i \;=$$

$$-br \bmod i \;=\; b(i-r) \bmod i.$$

Hence $Q' = (i-r) \bmod i,\;\; 2(i-r) \bmod i,\;\; 3(i-r) \bmod i,\; \ldots,\; (i-1)(i-r) \bmod i$, which gives

$$M(k,i) = \left\lfloor \frac{k}{i} \right\rfloor + M(i, i-r) = \left\lfloor \frac{k}{i} \right\rfloor + M(i,r).$$

Case 2: $gcd(k,i) = j > 1$. Then $M(k,i) = M(k/j, i/j)$ and this case is reduced to Case 1. □

We have bounded the number of left-to-right maxima of D_k for arbitrary $k > 0$. However, for D_k to be used in the Block-Write or AWT algorithms, *every* schedule of D_k must be a permutation. This can be achieved by assuming that $k = p$ is a prime (padding is used when necessary). (Recall that by Lemma 2.2 for any prime p, each schedule of D_p is a permutation.)

Theorem 3.1 gives a bound on the contention of D_p with respect to the identity permutation. The next theorem shows that the same bound applies to a family of adversarial patterns.

THEOREM 3.2. *For any prime* p, *for all* $\delta \in D_p$ *we have* $Cont(D_p, \delta) = L(p) = O(p \log^2 p)$.

PROOF. (Sketch) Follows from Lemma 2.2 and Theorem 3.1. □

Intuitively, the last theorem shows that if the order in which the blocks are first processed in the Block-Write algorithm is identical to the order established by the schedule of one of the processors, then the same bound holds for contention.

Of course, in order to bound the work of Block-Write or AWT (as Lemmata 2.1 and 2.3 teach us) we need to obtain a low bound on contention with respect to *any* permutation on $[p-1]$. Since Theorem 3.2 falls short of showing the bound with respect to all permutations, we conduct an experimental study that leads us to the main conjecture of this work.

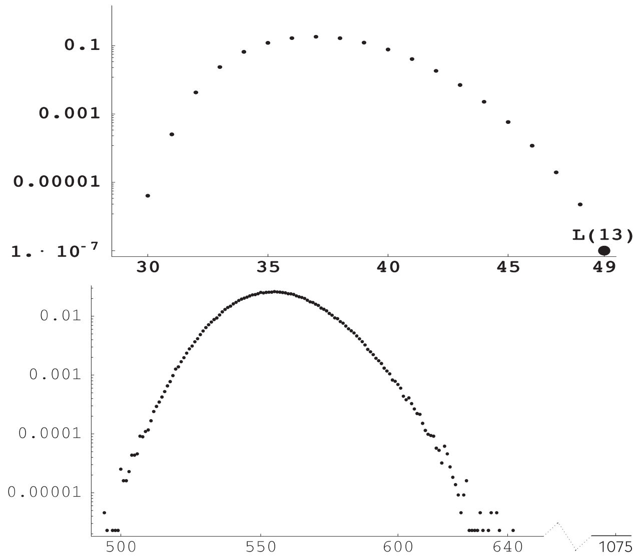

Figure 3: Plot of the fraction of permutations δ that yield given contention $Cont(D_p, \delta)$: (a) exhaustive search for D_{13}, $L(13) = 49$; (b) random sampling for D_{107} with values 0 discarded, $L(107) = 1075$.

3.2 The Conjecture and Experimental Support

First we present the results of an exhaustive study where we evaluate the contention of D_p with respect to all permutations on $[p-1]$. The exhaustive search is performed for each prime $p \leq 13$. For a given p we calculate the contention of D_p with respect to all permutations on $[p-1]$. The contention of D_p with respect to any permutation is *never* greater than $L(p)$ – the contention with respect to the identity permutation, see Table 1. For some permutations the contention is equal to $L(p)$. The number of such permutations, $|\{\delta : Cont(D_p, \delta) = L(p)\}|$, is given in the fourth column; the last column shows the ratio of the number to $(p-1)!$. Observe that the number of such permutations has to be at least $p-1$ (cf. Theorem 3.2). For the largest prime that we can evaluate given modest computing resources, $p = 13$, we plot the histogram (see Figure 3) of the fraction of permutations on [12] that yield a given number of left-to-right maxima.

p	$(p-1)!$	$L(p)$	equal to $L(p)$	ratio
2	1	1	1	1
3	2	3	2	1
5	24	9	8	0.33
7	720	17	12	0.016
11	3628800	37	60	0.000016
13	479001600	49	48	0.0000001
17	20922789888000	75	Still running	Unknown

Table 1: Summary of exhaustive search.

In the exhaustive search the contention of D_p with respect to any permutation is at most $L(p)$ and the ratio of permutations that yield contention $L(p)$ diminishes dramatically as p grows. These observations lead us to the following conjecture.

CONJECTURE 3.1. *WISH: $Cont(D_p) = L(p)$.*

We further support the conjecture with random sampling. We perform random sampling for primes from 17 to 107. For a given prime p we select uniformly $5 \cdot 10^9/p^2$ pseudorandom permutations δ and calculate contention of D_p with respect to δ. We observe that the contention is never greater than $L(p)$ (see Figure 3 for a plot of the probability mass function obtained in the simulation).

Next we show that we can efficiently construct a set of permutations with low contention as long as the conjecture is true. Assuming that primes are available the cost of constructing permutations D_p is completely amortized.

THEOREM 3.3. *Under WISH, for any prime p, $Cont(D_p) = \frac{3}{\pi^2} q \ln^2 q(1 + o(1)) = O(q \log^2 q)$ and each permutation can be constructed in $O(q)$ time on-line.*

4. COMPARISONS OF ALGORITHMS

In this section we apply the results from the previous section where we analyzed the contention of permutations in D_p. First we show an improved analysis of algorithm Y that uses n processors for $t = n^u$ tasks, $u \geq 2$. Under Conjecture 3.1 we show its work to be $O(t \log^2 t)$. Next we define a family of asynchronous *Write-All* algorithms for full range of processors $n \leq t$ with $O(t \cdot n^{\varepsilon})$ work contingent upon our conjecture. We show that our method yields a faster construction of $O(tn^{\varepsilon})$ *Write-All* algorithms than the construction given by Anderson and Woll [1]. We also compare our family against a construction of $O(tn^{\varepsilon})$-work *Write-All* algorithms induced by the construction of Naor and Roth [29]. Our approach yields more efficient *Write-All* algorithms with the same asymptotical complexity.

4.1 Efficiency of algorithm Y

In this section we bound the work of algorithm Y by applying the results from the previous section. This improves upon the claim in [20]. First we report a technical lemma. Assuming primes are not readily available, we show the complexity of finding them.

LEMMA 4.1. *For any $n > 0$ it is possible to find a prime p, $n < p \leq 2n$, in $O(n \log n)$ time.*

It is possible to give a tighter result and show that a needed prime is always in $[n, n + n^{11/20+\epsilon}]$, for any $\varepsilon > 0$, and big enough n [16]. However the lemma as stated is sufficient for our purposes. Assuming our Conjecture 3.1, using the fact that we can quickly find a prime, and using standard padding techniques we show that algorithm Y has $O(t \log^2 t)$ work, and its construction is amortized.

THEOREM 4.2. *With WISH, there is a deterministic Write-All algorithm with work $O(t \log^2 t)$ for n processors and $t = n^u$ memory cells, $u \geq 2$; the preprocessing time to construct this algorithm is $O(n \log n)$.*

The work of algorithm Y is greater than that of algorithm AW by the factor $\log t$ (t is the number of tasks), however its preprocessing is significantly lower than the preprocessing for algorithm AW. Recall that the preprocessing time for algorithm AW is exponential in n [1], while the preprocessing for Y is only $n \log n$ (or even 0 if the prime is given). This initial expense for Y is fully amortized.

4.2 Family $\mathrm{AWTD}_\varepsilon$ of $O(tn^\epsilon)$ work algorithms

We define the families $\mathrm{AWTD}_\varepsilon$ of *Write-All* algorithms indexed by $\epsilon > 0$. These algorithms are defined using algorithm AWT and the permutations studied in Section 3 as follows:

$$\mathrm{AWTD}_\varepsilon = \{\mathrm{AWT}(D_p) : -1 + \log_q Cont(D_p) \le \varepsilon\}.$$

Assuming Conjecture 3.1, by Theorem 3.3 for any $\varepsilon > 0$ we can find $p = q+1$ such that $\log_q(Cont(D_p)) \le 1+\varepsilon$. Hence $\mathrm{AWTD}_\varepsilon$ is nonempty for any $\varepsilon > 0$. By Lemma 2.3 any algorithm in $\mathrm{AWTD}_\varepsilon$ solves *Write-All* in $O(tn^\varepsilon)$ work.

4.3 Comparison of preprocessing for $\mathrm{AWTD}_\varepsilon$ and $\mathrm{AWTP}_\varepsilon$

Now we show that an algorithm from $\mathrm{AWTD}_\varepsilon$ can be instantiated quicker than any algorithm from $\mathrm{AWTP}_\varepsilon$. Under Conjecture 3.1 our method produces *Write-All* algorithms with the same work complexity faster. We first show how much preprocessing is required to produce any $O(tn^\varepsilon)$ algorithm in $\mathrm{AWTP}_\varepsilon$. Then we show how much preprocessing is sufficient to construct an $O(tn^\varepsilon)$ algorithm in $\mathrm{AWTD}_\varepsilon$.

LEMMA 4.3. *To obtain any $O(tn^\varepsilon)$ work algorithm in $AWTP_\epsilon$, $0 \le \epsilon \le 1$, the procedure presented in [1] requires $\Omega(e^{c^2/\varepsilon^2 - c/\varepsilon})$ work, $c = \ln 2$.*

PROOF. First we establish a lower bound on the value of q for a given $\varepsilon > 0$ that q must have so that $-1 + \log_q Cont(P) \le \varepsilon$. Observe that any list of q permutations must have contention at least $2q$. Hence $-1+\log_q Cont(P) \ge \log_q 2$. Therefore we must have $\log_q 2 \le \varepsilon$, and so $q \ge 2^{1/\varepsilon}$. The methodology presented in [1] does not offer a better bound on work than Lemma 2.3. Hence in order to get an $O(tn^\varepsilon)$ work algorithm from $\mathrm{AWTP}_\varepsilon$, we must have $q \ge 2^{1/\epsilon} = e^{(\ln 2)/\epsilon} \ge \frac{\ln 2}{\epsilon}$.

According to the procedure in [1], a set P of q permutations with contention at most $3qH_q$ is found by evaluating consecutive sets, each against all permutations on $[q]$. In the best case the first set has contention up to $3qH_q$, and so $\Omega(q!)$ work is needed to conduct the search.

Now we use Stirling's formula to derive a lower bound on work needed to construct any member of $\mathrm{AWTP}_\varepsilon$ that has $O(qtn^\varepsilon)$ work. We note that $q! \ge \left(\frac{q}{e}\right)^q \ge \left(\frac{e^{(\ln 2)/\epsilon}}{e}\right)^{\left(e^{(\ln 2)/\epsilon}\right)} \ge e^{((\ln 2)/\epsilon - 1)(\ln 2)/\epsilon}$. Hence the result follows. □

LEMMA 4.4. *Under WISH, for every $0 < \varepsilon < 1/e$ that is sufficiently small, an $O(tn^\varepsilon)$ work algorithm in $AWTD_\varepsilon$ can be constructed using $O(\frac{10}{\varepsilon}(\ln \frac{1}{\varepsilon})e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}})$ work.*

PROOF. By Lemma 2.3, and since D_p can be constructed on-line, it is enough to show that we can find a prime $p = q+1$ such that $-1+\log_q Cont(D_p) \le \varepsilon$ using $O(\frac{10}{\varepsilon}(\ln\frac{1}{\varepsilon})e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}})$ work.

By Bertrand's postulate, there is always a prime p, $2\lceil e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}}\rceil + 2 \ge p \ge \lceil e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}}\rceil + 1$. By Theorem 3.3, the contention of D_p for a big enough prime $p = q+1$ is at most $0.4 \cdot q\ln^2 q$. Using this bound on $Cont(D_p)$ we obtain $-1 + \log_q Cont(D_p) \le \frac{\ln 0.4 + 2\ln\ln q}{\ln q}$. Let $f(q) = \frac{\ln 0.4+2\ln\ln q}{\ln q}$. Observe that $q \ge e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}} \ge e^3$, and that beyond e^3 the function $f(\cdot)$ is decreasing. Hence $f(q) \le f(e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}}) = \frac{\varepsilon}{10\ln\frac{1}{\varepsilon}}\left(\ln 40 + 2\ln\frac{1}{\varepsilon} + 2\ln\ln\frac{1}{\varepsilon}\right) \le \varepsilon$. By Lemma 2.3, the AWT algorithm parameterized with D_p yields a *Write-All* algorithm with $O(qtn^\varepsilon)$ work. Using Lemma 4.1 we can find p in $O(\frac{10}{\varepsilon}(\ln\frac{1}{\varepsilon})e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}})$ work. □

We show that our algorithm needs significantly less preprocessing to achieve the same asymptotic work complexity. In the above lemmata we bound the preprocessing for algorithms in both families. Since $\lim_{\varepsilon\to 0^+} \frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}} \cdot e^{-(\ln^2 2)/\epsilon + (\ln 2)/\epsilon} = 0$, our approach yields an $O(tn^\varepsilon)$ algorithm faster.

THEOREM 4.5. *Under WISH, for sufficiently small $\varepsilon > 0$, constructing an algorithm in $AWTD_\varepsilon$ takes less work than constructing any algorithm in $AWTP_\varepsilon$ using method from [1].*

4.4 Comparison of work for $\mathrm{AWTD}_\varepsilon$ and $\mathrm{AWTN}_\varepsilon$

In this section we show that an algorithm in $\mathrm{AWTD}_\varepsilon$ has smaller constant hidden by the asymptotic analysis as compared to any algorithm in $\mathrm{AWTN}_\varepsilon$. First we show a lower bound on the size of permutations used by any algorithm in $\mathrm{AWTN}_\varepsilon$. The proof, which relies on bounds developed in [29] and [1], is provided in the appendix.

LEMMA 4.6. *To obtain any $O(tn^\varepsilon)$ work algorithm in $AWTN_\varepsilon$, the procedure and analysis presented in [29] and [1] require that the size q of the list of permutations used to instantiate the algorithm be $q \ge e^{1/\varepsilon^3}$.*

PROOF. The analysis presented in [29] bounds the contention of the list N of $q = m^{m!^m}$ permutations by $\log_q Cont(N) \le 1 + \frac{\ln 3 + \ln\ln m}{\ln m} + \epsilon(m)$. In order for this bound to be at most $1+\varepsilon$ we must have $m \ge e^{1/\varepsilon}$. Only then does the analysis of [1] guarantee that the corresponding algorithm AWT(N) has work $O(tn^\varepsilon)$. But then $q = (m^m)^{m!} \ge (m^m)^m \ge e^{1/\varepsilon^3}$. □

Now we are ready to compare the work of algorithms from $\mathrm{AWTD}_\varepsilon$ and $\mathrm{AWTN}_\varepsilon$. For a given $\varepsilon > 0$, by the proof of Lemma 4.4, and assuming that Conjecture 3.1 holds, we can instantiate an $O(e^{\frac{10}{\varepsilon}\ln\frac{1}{\varepsilon}}tn^\varepsilon)$ work *Write-All* algorithm. By Lemma 4.6 using methods from [1] and [29] yields an

$O(e^{1/\varepsilon^3} tn^\varepsilon)$ work algorithm. Hence for fixed big enough n and t, and small enough ε our approach yields a more efficient *Write-All* algorithm.

THEOREM 4.7. *Under WISH, for any $t \geq n$ big enough, and $\varepsilon > 0$ small enough, an algorithm in $AWTD_\varepsilon$ has smaller constant hidden by the asymptotic analysis than any algorithm from $AWTN_\varepsilon$ obtained using methods from [29] and [1].*

5. FUTURE WORK

In some cases the values of n and t are given up front, and the goal is to obtain the most efficient *Write-All* algorithm. We plan to investigate tradeoffs between q and ϵ to yield the lowest work algorithm for fixed n and t. We also plan to study the effect of constant on work complexity when comparing algorithm AWTD_ϵ, AWTP_ϵ, and AWTN_ϵ.

6. ACKNOWLEDGMENTS

We thank Alex Russell, Burkhard Englert, and Mikołaj Rotkiewicz for discussion, and anonymous SPAA reviewers for helpful comments.

7. REFERENCES

[1] Anderson, R.J., Woll, H.: Algorithms for the Certified Write-All Problem. SIAM Journal on Computing **26** (5) (1997) 1277–1283

[2] Aumann, Y., Rabin, M.O.: Clock Construction in Fully Asynchronous Parallel Systems and PRAM Simulation. Proc. of the 33rd IEEE Symp. on Foundations of Computer Science (1993) 147–156

[3] Aumann, Y., Kedem, Z.M., Palem, K.V., Rabin, M.O.: Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs. Proc. of the 34th IEEE Symp. on Foundations of Computer Science (1993) 271–280

[4] Buss, J., Kanellakis, P.C., Ragde, P.L., Shvartsman, A.A.: Parallel Algorithms with Processor Failures and Delays. J. of Algorithms **20** (1996) 45–86

[5] Chandrasekharan, K.: Introduction to Analytic Number Theory. Springer-Verlag (1968)

[6] Cole, R., Zajicek, O.: The APRAM: Incorporating Asynchrony into the PRAM Model. Proc. of the 1989 ACM Symp. on Parallel Algorithms and Architectures (1989) 170–178

[7] R. Cole and O. Zajicek: The Expected Advantage of Asynchrony. Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures (1990) 85–94

[8] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., van Eicken, T.: LogP: Towards a Realistic Model of Parallel Computation. 4th ACM PPOPP (1993) 1–12

[9] Dasgupta, P., Kedem, Z., Rabin, M.: Parallel Processing on Networks of Workstation: A Fault-Tolerant, High Performance Approach. Proc. of the International Conference on Distributed Computer Systems (1995) 467–474

[10] Eppstein, D., Galil, Z.: Parallel techniques for combinatorial computation. Annual Computer Science Review, Vol. 3 (1988) 233–83

[11] Fortune, S., Wyllie, J.: Parallelism in Random Access Machines Proc. the 10th ACM Symposium on Theory of Computing (1978) 114–118

[12] Gibbons, A., Spirakis, P. (Eds.): Lectures on Parallel Computation. Cambridge International Series on Parallel Computation: 4, Cambridge University Press (1993)

[13] Gibbons, P.: A More Practical PRAM Model. Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures (1989) 158–168

[14] Groote, J.F., Hesselink, W.H.: Synchronization-free parallel accessible hash-table. In preparation (1999)

[15] Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An algorithm for the asynchronous write-all problem based on process collision. Distributed Computing (2001)

[16] Guy, R.: Unsolved Problems in Number Theory. Springer-Verlag (1994)

[17] Hesselink, W.H., Groote, J.F.: Waitfree Distributed Memory Management by Create, and Read Until Deletion (CRUD). Technical report SEN-R9811, CWI, Amsterdam (1998)

[18] Hornick, S.W., Preparata, F.P.: Deterministic P-RAM: simulation with constant redundancy. Proc. of the 1989 ACM Symp. on Parallel Algorithms and Arch. (1989) 103–109

[19] Kanellakis, P.C., Shvartsman, A.A.: Efficient Parallel Algorithms Can Be Made Robust. Distributed Computing, vol. 5:4 (1992) 201–217 (Preliminary version: Proc. of the 8th ACM PODC (1989) 211–222)

[20] Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997)

[21] Karp, R.M., Ramachandran, V.: Survey of parallel algorithms for shared-memory machines. Handbook of Theoretical Computer Science, J. van Leeuwen, Ed., North-Holland (1990)

[22] Kedem, Z.M., Palem, K.V., Rabin, M.O., Raghunathan, A.: Efficient Program Transformations for Resilient Parallel Computation via Randomization. Proc. 24th ACM Symp. on Theory of Comp (1992) 306–318

[23] Kedem, Z.M., Palem, K.V., Spirakis, P.: Efficient Robust Parallel Computations. Proc. 22nd ACM Symp. on Theory of Computing (1990) 138–148

[24] Kontogiannis, S., Pantziou, G., Spirakis, P., Yung, M.: Dynamic-Fault-prone BSP: A Paradigm for Robust Computation in Changing Environments. Proc. of the 10th ACM Symp. on Parallel Algorithms and Architectures (1998) 37–46

[25] Knuth, D.E.: The Art of Computer Programming Vol. 2 (third edition). Addison-Wesley Pub Co. (1998)

[26] Knuth, D.E.: The Art of Computer Programming Vol. 3 (third edition). Addison-Wesley Pub Co. (1998)

[27] Martel, C., Park, A., Subramonian, R.: Work-optimal Asynchronous Algorithms for Shared Memory Parallel Computers. SIAM Journal on Computing, Vol. 21 (1992) 1070–1099

[28] Martel, C., Subramonian, R., Park, A.: Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs. Proc. 32d IEEE Symp. on Foundations of Computer Science (1990) 590–599

[29] Naor, J., Roth, R.M.: Constructions of Permutation Arrays for Certain Scheduling Cost Measures. Random Structures and Algorithms **6** (1) (1995) 39–50

[30] Nishimura, N.: Asynchronous Shared Memory Parallel Computation. Proc. 3rd ACM Symp. on Parallel Algor. and Architect. (1990) 76–84

[31] Papadimitriou C.H., Yannakakis, M.: Towards an Architecture-Independent Analysis of Parallel Algorithms. Proc. of the 20th Annual ACM Symp. on Theory of Computing (1988) 510–513

[32] Shvartsman, A.A.: Achieving Optimal CRCW PRAM Fault-Tolerance. Information Processing Letters, Vol. 39:2 (1991) 59–66

[33] Valiant, L.: General Purpose Parallel Architectures. Handbook of Theoretical Computer Science (ed. J. van Leeuwen), Vol. 1, North-Holland (1990)

[34] Valiant, L.: A Bridging Model for Parallel Computation. Communications of the ACM, Vol. 33:8 (1990) 103–111

[35] Yao, A., Knuth, D.: Analysis of the subtractive algorithm for greatest common divisors. Proceedings of National Academy of Sciences, USA, 72:4720-4722 (1975)